VM-MAD: A cloud/cluster software for service-oriented academic environments

Aleksiev, Tyanko ; Barkow-Oesterreicher, Simon ; Kunszt, Peter ; Maffioletti, Sergio ; Murri, Riccardo ; Panse, Christian

Abstract: The availability of powerful computing hardware in IaaS clouds makes cloud computing attractive also for computational workloads that were up to now almost exclusively run on HPC clusters. In this paper we present the VM-MAD Orchestrator software: an open source framework for cloudbursting Linux-based HPC clusters into IaaS clouds but also computational grids. The Orchestrator is completely modular, allowing flexible configurations of cloudbursting policies. It can be used with any batch system or cloud infrastructure, dynamically extending the cluster when needed. A distinctive feature of our framework is that the policies can be tested and tuned in a simulation mode based on historical or synthetic cluster accounting data. In the paper we also describe how the VM-MAD Orchestrator was used in a production environment at the Functional Genomics Center Zurich to speed up the analysis of mass spectrometry-based protein data by cloudbursting to the Amazon Elastic Compute Cloud. The advantages of this hybrid system are shown with a large evaluation run using about hundred large (EC2) nodes.

DOI: https://doi.org/10.1007/978-3-642-38750-0_34

Originally published at:
Aleksiev, Tyanko; Barkow-Oesterreicher, Simon; Kunszt, Peter; Maffioletti, Sergio; Murri, Riccardo; Panse, Christian (2013). VM-MAD: A cloud/cluster software for service-oriented academic environments. In: Supercomputing - 28th International Supercomputing Conference, ISC 2013, Leipzig, 16 June 2013 - 20 June 2013. Springer, 447-461.
DOI: https://doi.org/10.1007/978-3-642-38750-0_34

VM-MAD: a cloud/cluster software for service-oriented academic environments

Tyanko Aleksiev[2], Simon Barkow-Oesterreicher[1], Peter Kunszt[3], Sergio Maffioletti[2], Riccardo Murri[2], and Christian Panse[1]

[1] Functional Genomics Center Zürich
ETH Zürich / Universität Zürich
Winterthurerstrasse 190, CH-8006 Zürich, Switzerland
e-mail: cp@fgcz.ethz.ch, simon.barkow@fgcz.uzh.ch
[2] Grid Computing Competence Center
Universität Zürich
Winterthurerstrasse 190, CH-8006 Zürich, Switzerland
e-mail: tyanko.alexiev@gmail.com, sergio.maffioletti@gc3.uzh.ch, riccardo.murri@gmail.com
[3] SystemsX, ETH Zürich
Clausiusstrasse 45, CH-8092 Zürich, Switzerland
e-mail: peter.kunszt@systemsx.ch

Abstract. The availability of powerful computing hardware in IaaS clouds makes cloud computing attractive also for computational workloads that were up to now almost exclusively run on HPC clusters.
In this paper we present the VM-MAD *Orchestrator* software: an open source framework for cloudbursting Linux-based HPC clusters into IaaS clouds but also computational grids. The *Orchestrator* is completely modular, allowing flexible configurations of cloudbursting policies. It can be used with any batch system or cloud infrastructure, dynamically extending the cluster when needed. A distinctive feature of our framework is that the policies can be tested and tuned in a simulation mode based on historical or synthetic cluster accounting data.
In the paper we also describe how the VM-MAD *Orchestrator* was used in a production environment at the Functional Genomics Center Zurich to speed up the analysis of mass spectrometry-based protein data by cloudbursting to the Amazon Elastic Compute Cloud. The advantages of this hybrid system are shown with a large evaluation run using about hundred large Elastic Compute Cloud (EC2) nodes.

1 Introduction

Recent years have seen great advances in virtualization technologies, to the point that it is now possible to run computationally-heavy workloads on completely virtualized infrastructures. Starting with Amazon EC2, commodity on-demand virtualized compute infrastructures[4] have become affordable to anyone. They

[4] Commonly referred to as "Infrastructure-as-a-Service (IaaS) clouds".

include virtualized compute and storage hardware, dedicated networking and a software stack entirely under control of the end-user.

Therefore, the use of virtualized computational infrastructures has become very appealing to smaller research groups: it is now possible to access large computational resources without the need to buy and maintain a corresponding hardware infrastructure.

Today, emerging computational disciplines (e.g., Bioinformatics, Medical informatics) are showing usage patterns that do not fit well in the traditional High-Performance Computing (HPC) model of few individual jobs making use of the entire infrastructure through massively parallel programming. Their model is to submit a very large number of small jobs in bursts to analyze the relevant data, and then post-process the results to get a statistical overview or model prediction. Their need for computational resources in terms of CPU hours is similar to the massively parallel HPC use-cases but without the need for low-latency networks for MPI communication. HPC resource providers, who need to support such user communities with transient "peak" workloads, cannot afford to plan the infrastructure for peak usage, as it would be underutilized for most of the time. At the same time, they do not want to see a negative impact on the traditional HPC cluster users either. Therefore, exploitation of cloudbursting to IaaS clouds for HPC is interesting also to small and mid-sized facilities.

The term "cloudbursting" describes the ability of a local computational resource facility to dynamically add virtual machine instances from IaaS providers to their local resource, extending it in size elastically as needed. Cloudbursting improves application throughput and response time as seen by the user. It is an efficient technique for dynamic HPC resources expansion and peak workload offloading.

Cloudbursting also allows to add cluster nodes to the local resource that extends it with new abilities to the benefit of the users. For example, it is possible to extend the local cluster with virtual nodes enabling Hadoop workloads, or special GPU workloads that are not supported locally.

In this paper we present the Virtual Machines Management and Advanced Deployment (VM-MAD) *Orchestrator* software: an open source framework for cloudbursting Linux-based HPC clusters into IaaS clouds. The VM-MAD *Orchestrator* is completely modular, allowing flexible configurations of cloudbursting policies in the Python programming language. It can be used with any batch-queuing cluster system or cloud infrastructure, dynamically extending the cluster when needed. The policies can be tested and tuned by using the VM-MAD *Orchestrator* in simulation mode, based on historical or synthetic cluster accounting data.

The paper is organized as follows. We first discuss the design goals of the VM-MAD *Orchestrator* and the architecture we devised to implement them (Section 2). In section 3 we take a more in-depth look at the implementation and discuss how cloudbursting policies are configured in VM-MAD. As a real-world use case example, we report on the usage of the VM-MAD *Orchestrator* to run some special ensemble jobs on the bioinformatics cluster at the Functional Ge-

nomics Center Zurich (Section 4). Finally, we survey similar and concurrently-developed solutions (Section 5) and outline some conclusions and possible future developments (Section 6).

1.1 List of acronyms

API	Application Programming Interface
CPU	Central Processing Unit
EC2	Elastic Compute Cloud
ECU	EC2 Compute Unit
ETHZ	*Eidgenössische Technische Hochschule Zürich*, Swiss Federal Institute of Technology Zurich
FGCZ	Functional Genomics Center Zurich
HPC	High-Performance Computing
IaaS	Infrastructure-as-a-Service
IBM	International Business Machines
LSF	Load Sharing Facility (a batch-queuing system)
SMSCG	Swiss Multi-Science Computational Grid
UZH	University of Zurich
VM	Virtual Machine
VM-MAD	Virtual Machines Management and Advanced Deployment (the project described in this paper)
VPN	Virtual Private Network

2 Overall design and architecture

The stated goal of the VM-MAD project was to build a stable software service that could be used on existing production-grade HPC cluster infrastructures to dynamically add computing power during peak loads, and to automatically revert to using only local processing facilities when the "rush hour" is over. This elastic "cloudbursting" feature should have as little impact as possible on the current usage patterns of HPC clusters; ideally, nothing should change in the HPC users' experience but the system would automatically launch cloud-based Virtual Machines (VMs) and schedule jobs that would otherwise not be possible or take too much time or resources out of the cluster.

2.1 Implementation requirements

Early in the development process, we realized that achieving these goals entails dealing with large heterogeneity.

First of all, we would need to accommodate different batch-queuing systems, even if we are restricting ourselves to the HPC clusters in use at the University of Zurich (UZH) and the *Eidgenössische Technische Hochschule Zürich* (ETHZ). While they all share the same workflow and interaction models, details of the submission Application Programming Interface (API) vary greatly. This ruled

out the possibility of implementing the VM-MAD cloudbursting software as an extension package for a particular batch system implementation. Instead we decided to interact with the batch system via the available command-line tools.

The second very important consideration was the actual definition of what is meant by "peak load", i.e., under what conditions the computing power should be extended using VM instances from the cloud, and what kinds of jobs can be run on the elastic part of the infrastructure. Defining peak load is a subtle matter of local policy. Any choice of a domain-specific language would have constrained the range of supported policies and therefore limited the applicability of the VM-MAD cloudbursting system. We chose instead to allow the definition of the local policy as a set of functions written in the Python programming language [17]: now a decision can be taken on the basis of *all* the data available to the cloudbursting software (see details in Section 3).

Finally, the "cloud" ecosystem is currently very dynamic. For our software to be useful even just in the next few years, it needs to be able to interface to different IaaS cloud infrastructures.

Based on these requirements we opted for a completely modular architecture: the VM-MAD software is a *framework* for building cloudbursting scripts, perfectly adapted to the peculiarities of each HPC installation. It is not a ready-made add-on for a particular product that a systems administrator can deploy with just a few touches to a configuration file.[5]

2.2 Architecture overview

Our solution to the "cloudbursting" problem, as outlined in the previous sections, is to build a tool which can be run as an add-on to existing batch systems. Our "Orchestrator" software implements the additional services needed to link the batch system with an elastic IaaS infrastructure. It' runs in the background[6] and performs the following tasks:

a. Monitor the jobs queued in the batch system, and select those that could run on a cloud-based VMs;
b. Start and shut down VM instances;
c. Add and remove VMs compute nodes to and from the cluster.

It is very important to remark that points *a.* and *b.* involve taking decisions according to *configurable policies and metrics*. The cluster system administrator is responsible for these policies and metrics.

Figure 1 shows the interaction of the software components involved in a cloudbursting scenario under control of a VM-MAD Orchestrator.

[5] This is similar to the current situation for the HPC scheduling softwares: low-maintenance schedulers are also limited in configurability and functionality, while those that are flexible enough to implement complex policies are often custom-built or have a rich and detailed configuration language.
[6] It is a "daemon" in the UNIX terminology.

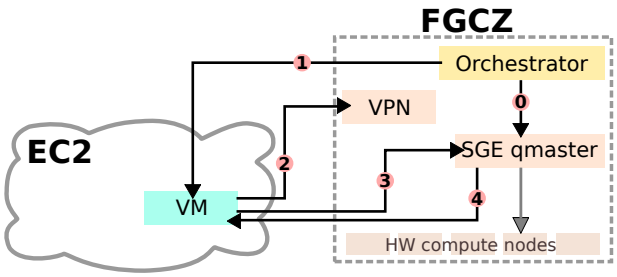

Fig. 1. Interaction of parts in a cloudbursting scenario. *(0)* The Orchestrator monitors the batch system state and determines when a new compute node is needed. *(1)* A new VM is started. *(2)* The VM connects back to the batch system network via VPN. *(3)* The VM is added to the cluster as a compute node. *(4)* The batch system can now start jobs on the VM.

(0) The Orchestrator monitors the batch system state and determines that —by the local policy definition— a new compute node is needed. (For example, the number of queued jobs that could be executed in a cloud-based VM exceeds a certain threshold.)
(1) The Orchestrator consults the cloud state and the local policy, and determines that the current set of cloud-based resources is insufficient. It therefore contacts the cloud provider via its network API and starts a new VM.
(2) The new VM connects back to the batch system network via a VPN. This requires that the VM image has been previously prepared by the cluster systems administrator: it should contain a the portion of the cluster execution environment that is necessary for running jobs destined to the cloud and the preconfigured VPN software to connect back to the "home" network.
(3) The Orchestrator adds the new VM to the cluster as a compute node, reconfiguring the batch system scheduler on the fly. All properties of this node are registered with the scheduler and jobs requesting those properties can be scheduled on the new cloud-based nodes.
(4) The batch system scheduler can now start jobs on the VM. It should be noted that the Orchestrator has a passive role with regards to scheduling computational jobs in the cloud: all it does is to start new VMs that satisfy the job requirements, and lets the batch system scheduler use those for actually running a job.
(5) When the Orchestrator detects that the amount of cloud-based resources exceeds the current needs (as defined by local policy), it shuts down the unneeded VMs.

3 Implementation overview

The VM-MAD cloudbursting framework is implemented as a library package written in the Python [17] programming language. The code is written in an object-oriented style; the basic components of the framework are Python classes.

The *Orchestrator* object is the core of the framework: it implements the main loop and performs housekeeping of the shared data structures. The *Orchestrator* is a singleton: only one single instance should be monitoring a given batch system. An *Orchestrator* instance must be adapted to the cluster setup by initializing it with a *SchedInfo* and a *Provider* instance.

SchedInfo objects are responsible for interacting with the batch system scheduler, especially for gathering information about the running/queued jobs and the available compute nodes. New batch-queuing systems can be supported by creating an appropriate *SchedInfo* subclass.

Provider objects are responsible for interacting with a remote IaaS cloud system and starting/stopping virtual machines.

The cloudbursting policy is defined by subclassing the *Orchestrator* object and overriding well-defined methods that decide whether a job is a candidate for cloud execution, or what type of virtual machine should be started.

It should be noted that this simple component architecture allows a great deal of flexibility: for instance, a *Provider* instance needs not interface to a cloud provider, but can also request nodes from a peer cluster or Grid infrastructure.[7] Likewise, the *SchedInfo* component does not need to read information from a live batch system: the standard VM-MAD software distribution includes components for replaying job information from a batch system accounting file, which can be used for simulating the effect of cloudbursting policies over historical data, see section 4.3. It also includes components to generate random workloads to be used for testing of the system and available infrastructure.

3.1 Policy definition

"Orchestrator policies" are criteria that govern decisions on whether:

1. a given job can run on cloud-based virtualized hardware;
2. a new VMs should be started to extend the current virtualized computational resource pool;
3. a running VMs should be stopped, shrinking the current virtualized resource pool.

For each of these decisions, a method is provided in the *Orchestrator* class that should return a *True/False* value based on the evaluation of available data. Systems administrators should override the default implementation to implement their chosen criteria.

Example: Policy on jobs eligible to run on virtualized hardware The decision whether a certain job can run on cloud-based resources is taken by the *is_cloud_candidate* method. This method is called once for each new job that

[7] This has actually been done in the course of the VM-MAD benchmarks, by starting VMs [14,3] on the Swiss Multi-Science Computational Grid (SMSCG) [18] computational grid infrastructure.

appears in the batch system queues and returns *True* if that job is eligible for cloudbursting. The default implementation always returns *False*, so that no job accidentally triggers the spawning of cloud-based VMs.

For example, the following code would implement a policy where only jobs that have been submitted to a special "cloud" queue trigger cloudbursting of compute resources:

```
1 def is_cloud_candidate(self, job):
2   return (job.queue == 'cloud.q')
```

The *job* record passed as argument to the *is_cloud_candidate* method contains all the information that the batch system scheduler provides via its queue-listing command (e.g., `qstat` on Sun/Oracle Grid Engine).

Example: Policy on starting new compute resources The decision on whether new cloud-based resources should be requested is taken by the *is_new_vm_needed* method. This method is called at each iteration of the Orchestrator's main loop. It has access to all the internal data structures, in particular the list of jobs eligible to run on cloud-based hardware (*self.candidates*) and the list of cloud-based VMs that have already been started by the Orchestrator (*self.vms*). By default, this method always returns *False*, so that cloud-based VMs are never spawned; this is a safety measure to avoid that non-configured Orchestrators start spawning VMs: since usage of cloud-based resources usually comes at a cost, it is entirely the administrator's task to decide when and how to initiate cloudbursting.

For example, the following code implements a policy where new cloud-based VMs are started if the number of queued candidate jobs is greater than double the number of VMs required to run them:

```
1 def is_new_vm_needed(self):
2   if len(self.candidates) > 2*len(self.vms):
3     return True
4   else:
5     return False
```

Example: Policy on stopping cloud-based compute resources At every iteration of the Orchestrator's main loop, a decision will also be taken on whether an idle VM (i.e., one that is not currently running any job) should be stopped. Since booting a cloud-based VM can take up to a few minutes' time, and many cloud infrastructure bill usage in hourly increments, it makes sense to try to reuse already-started VMs instead of starting new ones. The *can_vm_be_stopped* method is there exactly for this purpose: change the default Orchestrator behavior, which is to stop a VM as soon as it turns idle.

For example, the following code implements a policy where a VM is allowed to be idle for 10 minutes before it is stopped by the Orchestrator:

```
def can_vm_be_stopped(self, vm):
  TIMEOUT = 10*60 # 10 minutes
  if vm.last_idle > TIMEOUT:
    return True
  else:
    return False
```

4 Application and Testing

For testing we have chosen an area we have a lot of expertise in. Identifying proteins in a biological sample with the help of large computer systems is a common application in the life sciences which behaviour is well studied so that we have enough experience with all parameters and configuration details, e.g. memory consumption, input-output, and stability.

4.1 Test case: Analyzing mass spectrometric related protein data

The processing of mass spectrometry data can be challenging as it involves several computationally demanding algorithmic steps. Examples are the peptide spectrum assignment of mass spectrometry data to identify proteins in a biological sample, as well as the detection and identification of post-translational modifications of proteins. Both tasks can be computed simultaneously and can easily occupy hundreds of Central Processing Units (CPUs) for several days.

With every new mass spectrometer, the amount of measured data increases and the local computing infrastructures would need to be extended accordingly. However, these computing resources are only needed for a short period of time. The computation demand varies widely with the actual measurement type and the corresponding data set size. To be able to meet also larger use-cases, the available local cluster would need to be very large and powerful, but then it would be mostly under-utilized. Therefore such large use-cases are not feasible as currently the capacity cannot be extended on demand.

Large-scale so-called "shotgun experiments" with complex samples from, e.g., human or fruit fly involve about ten thousand proteins. The peptide spectrum matches for our test were computed with the SEQUEST and OMSSA search algorithm [9,10]. To benchmark the VM-MAD Orchestrator, we have run the search on both the local HPC cluster facility at the UZH, as well as on the Amazon EC2 Cloud computing resources in the Amazon region US-East. To avoid denial of service like failures on the cloud system, e.g., during file server and authentication operations we started our virtual machines in a staggered manner with a delay of 60 seconds. In order to avoid problems like hanging processes, that might be caused by a high latency of the network connection (e.g. for accessing a network filesystem), each of our compute jobs is responsible for dealing with its own input and output data.

4.2 Effectiveness: Benchmark of a real world data set

As a test data set we used a large scale proteomics *Drosophila* (fruit fly) experiment [4] consisting of 1800 (LC)-MS/MS runs, having a peptide mass window of 3 Dalton, 8474960 tandem mass spectra, 498000 redundant peptides, 72281 distinct peptides, and 9124 proteins. The data volume is approximately 0.3TB split into 1800 jobs. The whole experiment data and the graphics are included in the *cloudUtil* R-package [16]. In our benchmark we compare three compute systems: the small cluster at the FGCZ consisting of around 100 CPUs, a larger system as part of the Schroedinger cluster of the UZH, and a virtual cluster on Amazon EC2. For benchmarking we recorded network bandwidth, CPU performance (compute time) and robustness on all systems. An overview of the experiment and a relative comparison of each compute job can be seen in the utilization plot on Figure 3.

The box plots [5] in Figure 4 show a comparison of the run time and the network throughput on three compute systems having two repetitions. One EC2 Compute Unit (ECU) provides the equivalent CPU capacity of a 1.0-1.2 GHz 2007 Opteron or 2007 Xeon processor. The FGCZ cluster is based on Intel Xeon CPU E5450 3.0GHz and the UZH cluster is based on Intel Xeon CPU 5500.

4.3 Simulation of LRMS accounting data

For demonstrating the effectiveness of the Orchestrator software and to study the behavior of the Orchestrator policies for different hardware scenarios (i.e. number of nodes) and different accounting data we have implemented a simulation mode policy. If this policy is used the *Orchestrator* takes the batch-system accounting information and the cloud configuration file as input. The accounting file of the LRMS contains the ordered start time-stamps of every compute job and its corresponding run time. The configuration parameters are the time step argument (in seconds), the start time of the simulation, the maximum number of available hosts. Instead of orchestrating real nodes the Orchestrator writes all decisions about starting or stopping virtual machines to an output file. The visualization in Figure 5 shows the simulated state of the LRMS queue and the status of the VMs over time for different simulation runs. In particular, the plot on the bottom of Figure 5 displays the output of the following command line:

$$\begin{aligned} &\textit{simul.py} \; --\textit{time-interval}\; 30 \; --\textit{start-time}\; \text{'2008-12-16\_02:13:50\_CET'} \; \backslash \\ &\qquad --\textit{max-vms}\; 512 \; --\textit{cluster-size}\; 100 \; --\textit{csv-file}\; \textit{accounting.csv} \end{aligned}$$

The simulation mode can also be used for determining the optimal number of compute nodes for a given task; see. e.g., Figure 5.

5 Related work

Cloudbursting, as a compute model where local resources elastically allocate cloud instances for improving application throughput/response time, was first

proposed by Amazon's Jeff Barr [7]. There is a variety of different mechanisms for cloudbursting an on-premise computational cluster to an external cloud provider: the most common derives from the HTCondor glide-in model [8] that is used to add a machine running on an external provider to an existing HTCondor pool. HTCondor glide-in configures a remote resource such that it reports to and joins the local HTCondor pool. This is the technology used for example by CycleComputing.com.

Inspired by this model, workload management systems that do support cloudbursting, like Sun/Oracle Grid Engine [19], Moab [13], or the HTCondor CloudScheduler [6], allow to start a pre-configured virtual instance, that can reside on an external cloud provider, and let it join the pool of resources they control. While VM-MAD takes an open approach in providing cloudbursting capabilities that could be adapted to virtually any workload management system thanks to its plug-in based approach, Grid Engine and Moab do provide a vendor-specific solution based on policies and configurations that cannot be applied nor ported to other similar systems.

Another approach in supporting cloudbursting is provided by the Multi-Cluster [12] solution from IBM's Platform Computing. An existing on-premise Load Sharing Facility (LSF)-controlled cluster could be extended by starting an entire LSF cluster on a cloud provider and use Multi-Cluster to federate them The main limitation of these approaches is the lack of an automatic system to start and control an LSF cluster on a cloud provider.

In terms of cloudbursting out of applications, Software-as-a-Service solutions make use of IaaS clouds to assure their workloads are scaled properly. An example in the life science domain is the Galaxy CloudMan project [1,2]. Here the Galaxy portal makes use of cloud resources to extend the support for selected computational workflows. The CloudMan system, that is tightly coupled to the Galaxy portal, provides a pre-selected set of tools and services as well as the possibility of deploying own software tools and integrate them through a web interface. While Galaxy targets the sequencing community, with ProteoCloud [15], there exists also a cloud computing pipeline for proteomics applications but it does not feature automatic cloudburst functionality.

In contrast to the these specialized frameworks, our approach is not limited to life-science applications. Any community-specific portal that is already capable of using on-premise computational clusters could seamlessly profit from clouds by deploying the VM-MAD *Orchestrator*.

An example of cloudbursting from an on-premise cloud infrastructure to an external provider is brought by the Seagull project [11]. Seagull dynamically decides which running applications can be moved from the on-premise cloud infrastructure to the configured external provider, using an Intelligent Placement module based on a placement algorithm that picks those applications to move that free up the most units of local resources relative to their cost of running in the cloud using a pre-defined cost function. To reduce cloudbursting latency (due to the copying of the disk image corresponding to the selected running application), Seagull performs pre-copying by transferring an incremen-

tal snapshot of a virtual machine's disk-state to the cloud. Seagull focuses on cloud-to-cloud cloudbursting features, whereas VM-MAD allows to cloudburst a batch-controlled computational cluster; Seagull takes autonomous decisions on what running applications to migrate live; on the other hand, VM-MAD has a simple policy module to determine whether to launch new appliances on the connected cloud provider.

6 Conclusions and Future Work

In this work we have described the architecture, the implementation, and an application use case of the *Orchestrator* cloudbursting software framework developed by the the VM-MAD project. The *Orchestrator* allows an existing compute cluster to be extended, burst into the cloud based on a highly configurable set of local policies. We have successfully extended local clusters with Amazon EC2 instances. In the future we also want to run Hadoop applications on the virtual part of the cluster, enabling MapReduce applications for our local users. Also, connecting to non-public clouds, e.g., in-house OpenStack IaaS is possible, as well as to extend to other batch cluster systems.

The *Orchestrator* is a modular framework that can be interfaced with any batch-queuing system and IaaS cloud infrastructure. An interesting consequence of this modularity is that the *Orchestrator* can also be run in *simulation mode*, to allow testing cloudbursting policies against historical accounting data and evaluate the most cost-effective one. We will try to optimize the usage of historical data in the future to suggest good predefined policies to system administrators.

As a test and benchmark, we have used the VM-MAD *Orchestrator* for reprocessing a large set of proteomics data; the performance data collected show that commercial IaaS clouds can already deliver computational and network performance comparable to what is offered by a small in-house cluster, and are thus suitable for offloading peak computational workloads. We will also use the same concept with other workloads, like computational chemistry and structural biology, but also in the domains of geography and finance.

References

1. E. Afgan, D. Baker, N. Coraor, B. Chapman, A. Nekrutenko, and J. Taylor. Galaxy CloudMan: delivering cloud compute clusters. *BMC Bioinformatics Supplements*, 12:S4, 2010.
2. E. Afgan, D. Baker, t. G. Team, A. Nekrutenko, and J. Taylor. A reference model for deploying applications in virtualized environments. *Concurrency Computat.: Pract. Exper*, 24:1349–1361, 2012.
3. AppPot. `http://apppot.googlecode.com/`, January 2013.
4. Erich Brunner et al. A high-quality catalog of the drosophila melanogaster proteome. *Nature Biotechnology*, 25(5):576–583, April 2007.
5. W. S. Cleveland. *Visualizing Data*. Hobart Press, Summit, New Jersey, U.S.A., 1993.

6. Cloud scheduler — your htc jobs on the cloud. http://cloudscheduler.org, January 2013.
7. Cloudbursting, hybrid application hosting. http://aws.typepad.com/aws/2008/08/cloudbursting-.html, 2008.
8. HTCondor glide-in. http://research.cs.wisc.edu/htcondor/manual/v7.8/5_4Glidein.html.
9. Jimmy K. Eng, Ashley L. McCormack, and John R. Yates III. An approach to correlate tandem mass spectral data of peptides with amino acid sequences in a protein database. *Journal of the American Society for Mass Spectrometry*, 5, 1994.
10. L. Y. Geer, S. P. Markey, J. A. Kowalak, L. Wagner, M. Xu, D. M. Maynard, X. Yang, W. Shi, and S. H. Bryant. Open Mass Spectrometry Search Algorithm. *eprint arXiv:q-bio/0406002*, June 2004.
11. Tian Guo, Upendra Sharma, Timothy Wood, Sambit Sahu, and Prashant Shenoy. Seagull: intelligent cloud bursting for enterprise applications. In *Proceedings of the 2012 USENIX conference on Annual Technical Conference*, USENIX ATC'12, pages 33–33, Berkeley, CA, USA, 2012. USENIX Association.
12. IBM Platform Computing - Multicluster. http://www-03.ibm.com/systems/technicalcomputing/platformcomputing/products/symphony/index.html.
13. Moab Cloud solution. http://www.adaptivecomputing.com/home/cloud/.
14. Riccardo Murri and Sergio Maffioletti. AppPot: bridging the Grid and Cloud worlds. In *EGI Community Forum 2012*. PoS(EGICF12-EMITC2)004. Available online at: http://pos.sissa.it/.
15. Thilo Muth, Julian Peters, Jonathan Blackburn, Erdmann Rapp, and Lennart Martens. ProteoCloud: A full-featured open source proteomics cloud computing pipeline. *Journal of Proteomics*, January 2013.
16. Christian Panse and Ermir Qeli. *cloudUtil: Cloud Util Plots*, 2012. R package version 0.1.9, http://CRAN.R-project.org/package=cloudUtil.
17. Python Programming Language — Official Website. http://www.python.org/, January 2013.
18. Swiss multi-science computing grid. http://www.smscg.ch/, January 2013.
19. Sun/Oracle Grid Engine Cloud. http://www.oracle.com/technetwork/oem/cloud-mgmt/index.html.

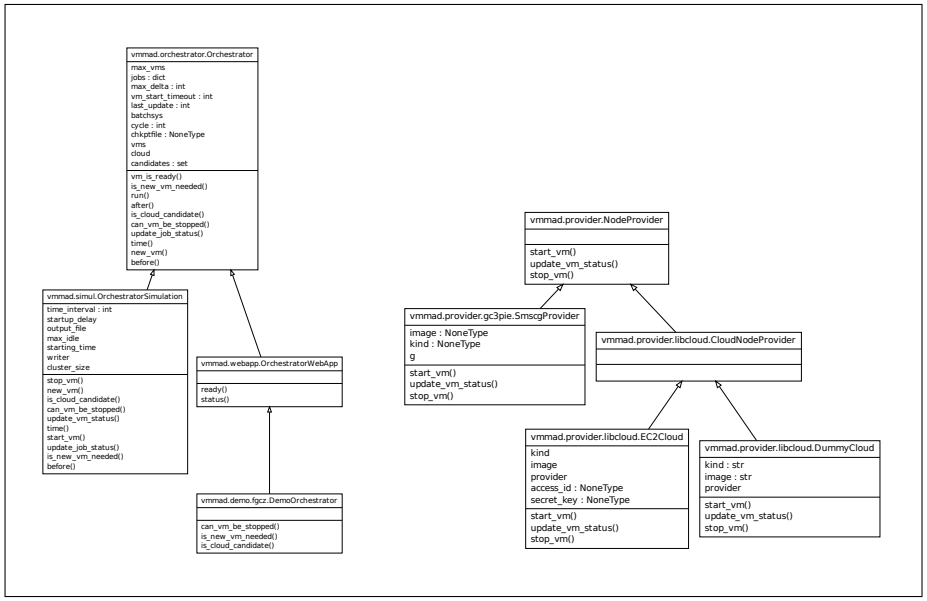

Fig. 2. *Left:* UML class diagram of the *Orchestrator* and its derived classes. The root of the hierarchy is the `vmmad.orchestrator.Orchestrator` class, which implements the main daemon loop and the core infrastructure for the VM-MAD functionality. Two derived classes are shown: the `vmmad.simul.OrchestratorSimulation` class is used to simulate running the VM-MAD software on historical accounting data; the `vmmad.demo.fgcz.DemoOrchestrator` is a an actual implementation of the VM-MAD Orchestrator for use on the FGCZ cluster. Note that `vmmad.demo.fgcz.DemoOrchestrator` derives from `vmmad.orchestrator.Orchestrator` through `vmmad.webapp.OrchestratorWebApp`, which implements a web interface for *Orchestrator* status reporting. *Right:* UML class diagram of the cloud interface classes. The root class `vmmad.provider.NodeProvider` defines the programming interface to which other classes must conform. Classes in the `vmmad.provider.libcloud` package implement interfaces to different IaaS cloud stacks using the Apache LibCloud library. The `vmmad.provider.gc3pie.SmscgProvider` draws nodes from clusters participating in the SMSCG computational grid infrastructure; it is an example of how VM-MAD can be interfaced to non-cloud infrastructures.

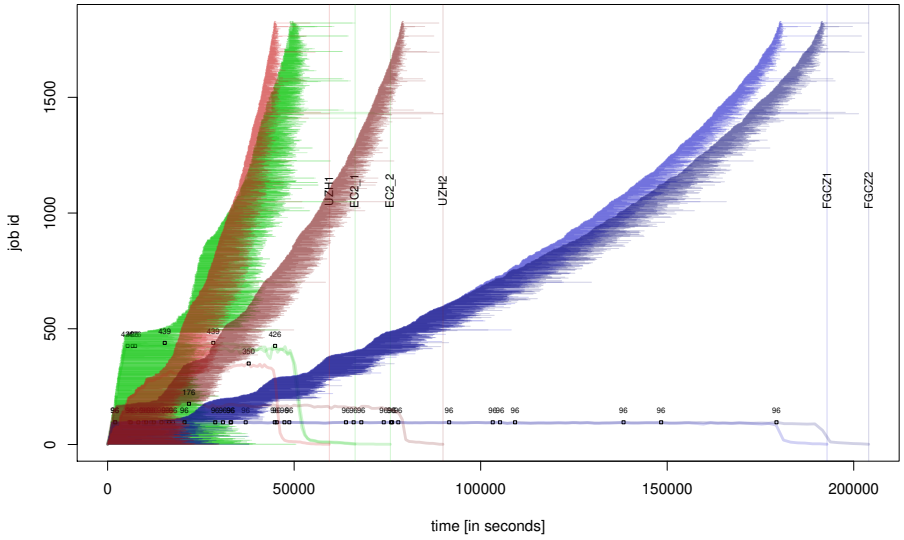

Fig. 3. Overview of the benchmark. On the utilization graph each horizontal line indicates the start and end of each job. The graph shows that the lines for the two jobs runs on the cloud (green) have almost the same length (we cannot distinguish 2 green branches) while the running times on the cluster nodes (red and blue) differ more significantly (the two repetitions are clearly distinguishable). This can be explained by the variable queue status of the cluster nodes because of other users using the cluster at the same time. Also, it takes much longer to run through all jobs on the limited FGCZ cluster (blue). The lines in the lower part of the graphic show the total number of concurrently running jobs. The squares on those lines indicate the maxima on the respective system.

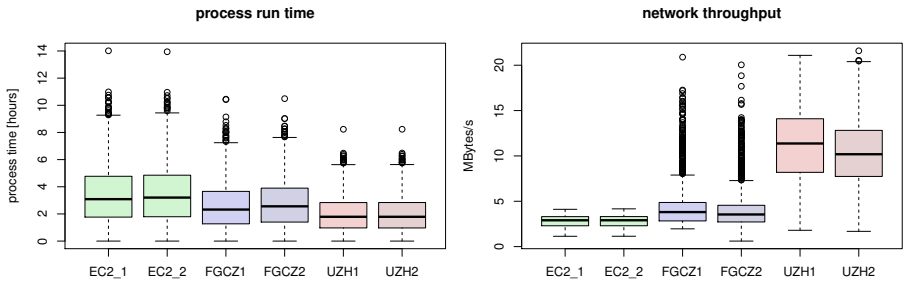

Fig. 4. Comparison run time and copy input I/O – The box plots [5] display the job run time distributions of the two repetitions of all three compute systems (left) and the copy I/O network throughput (right).

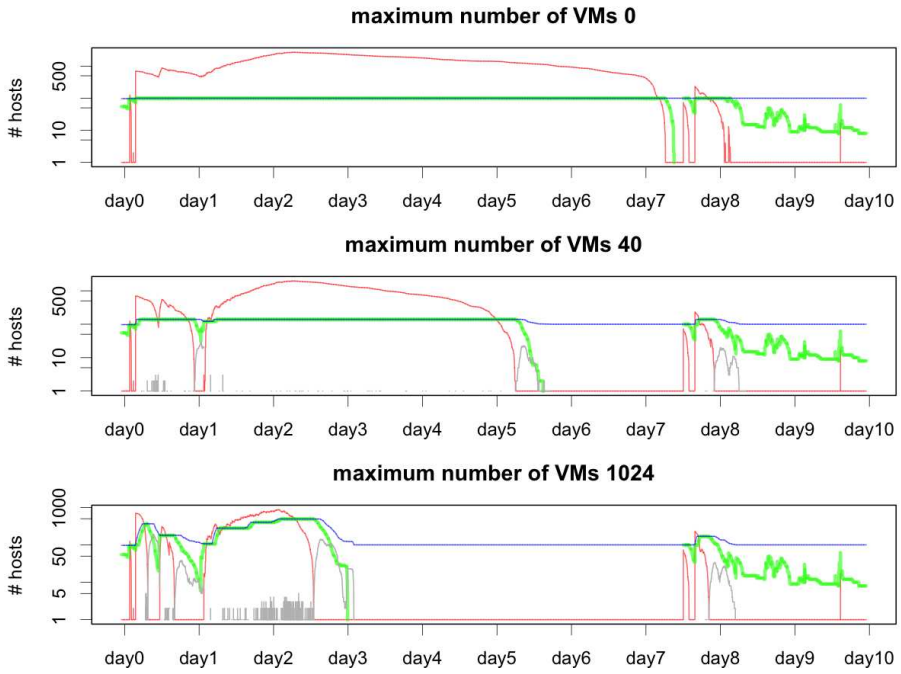

Fig. 5. The graphics show the simulation of 10 days of FGCZ batch cluster accounting data. The vertical axis showing the number of hosts/jobs is $\log_{10}$-scaled. The colored lines have the following meaning: *(red)* pending jobs; *(green)* running jobs; *(blue)* available nodes to the cluster (100 plus VMs); *(grey)* idle VMs. The upper simulation run corresponds to the FGCZ setup of 100 CPUs. For the simulation depicted in the lower graphic we added on demand up to 40 and 1024 VMs. For the computation we have submitted the described proteomics data set. It can be seen that with increasing number of VMs the overall compute time can be reduced to several days.